TR-103-1

FEASIBILITY OF SOFTWARE BUILT-IN TEST FOR SDI APPLICATIONS

August 1987

Final Report
Covering Period of Performance: 2/9/87 - 7/31/87

Contract No. N00014-87-C-0134

Prepared For:

Strategic Defense Initiative Organization
Office of Innovative Science and Technology
Washington, D.C. 20301-7100

Office of Naval Research
Department of the Navy
Arlington, VA 22217-5000

Distribution: Unlimited

Prepared by:
Richard D. Healy

Atlantis Research Group, Inc.
Peabody, MA 01960

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A185 750

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Distribution Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) TR-103-1	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Atlantis Research Group 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) One Intercontinental Way Peabody, MA 01960-3832	7b. ADDRESS (City, State, and ZIP Code) 800 N. Quincy Street Arlington, VA 22217-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Strategic Defense Initiative Organization 8b. OFFICE SYMBOL (If applicable) SDIO/IST	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER Cont. No. N00014-87-C-0134
8c. ADDRESS (City, State, and ZIP Code) The Pentagon Washington, DC 20301-7100	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 63221C; PROJECT NO. SBIR,FY86; TASK NO. SDIO; WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Feasibility of Software Built-In Test for SDI Applications

12. PERSONAL AUTHOR(S)
Richard D. Healy

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final Report	FROM 870209 TO 870731	870815	49

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
012	05	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Software, Built-In Test, Real Time

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Software Built-In Test (BIT) is a design technique for collecting information from operational software that will assist in identifying differences between the real Operating Environment and either the Design or Test Environments. The BIT senses and indicates where the software is operating in "new" or "overloaded" environmental conditions and may, therefore, be more likely to fail. (This anomalous situation may be the result of either hardware failure or software design error.) The technical challenge is to incorporate the large number of relatively simple BIT tests into the fault-tolerant and continuously operating environment likely to characterize a solution to the battle management portion of the SDI mission. The management challenge is to provide these technical assists in such a way that they can be implemented in operational software with a minimal increase in software development time; it is then reasonable to expect that BIT will not shift from a hard requirement

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. Neil Gerr	22b. TELEPHONE (Include Area Code) (202) 696-4321 22c. OFFICE SYMBOL 1111SP

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. continued -

to a nice-to-have feature as schedule pressures potentially impact development. Our approach overcomes the management problem by providing a standard set of tools for use within the software development environment which will implement BIT with a minimum amount of programmer action.

We have assembled standard BIT tests and tools into four general categories for installation within either a compiler or preprocessor so that very little programmer effort is required. These four categories, distilled from numerous application- and computer-specific candidates, are: event counters, rate samplers, range checkers, and data samplers. They are implemented as preprocessor instructions so that the programmer need only identify the variable(s) to be monitored, type of BIT, and the limits and priorities at the instrumentation location. The tool then produces the necessary source-code (which implements the BIT, interacts with the BIT database, prepares and dispatches messages, and resumes program execution).

Automating implementation relieves a particularly effective argument against the BIT process: that it takes too long to implement special tests. Moreover, we have developed a procedure for "training" the algorithm during testing and/or non-crisis modes of operation in order to permit evolution without re-coding and/or re-compiling the software. We believe that this will be most useful in the battle management arena, since it can be very difficult to anticipate the detailed evolution of command and control systems during the design phase of the initial phases of the software.

Successful completion of this research will provide a family of tools for assisting the battle manager in anticipating situations where system overload caused by design inconsistency is about to impact performance. This approach should provide a method of precluding or alleviating significant operational impact from a software fault that might otherwise cause anomalous system performance.

Executive Overview

This report presents the results of a six-month investigation of the application of the concept of Software Built-in Test (BIT) to systems that might be useful in achieving the objectives of the Strategic Defense Initiative Organization (SDIO). It was sponsored under the DoD Small Business Innovative Research (SBIR) Program and was administered by the Office of Naval Research (ONR) under Contract No. N00014-87-C-0134. The results reported herein are for Phase I, feasibility evaluation, of the SBIR effort.

Atlantis believes that the results demonstrate that significant additional resources should be applied to further refining the Software BIT concept. In particular, three features of the methodology (as refined to date) may be significant in the development of practical systems for SDIO:

- An automated tool can be used to generate the code needed to support Software BIT within the fault-tolerant, continuously operating environment likely to characterize the SDI battle management arena

- Software BIT can be implemented in a manner to permit "learning" during non-crisis operation as an aid to setting correct thresholds

- Separation of the detection and reporting features of the concept.

Additionally, we have identified particular implementation-specific features of Software BIT that will be important as the concept evolves and is implemented. Where possible, Ada[1] constructs have been considered.

While it is important to identify direct applicability for the concept, it is equally important to identify areas where the concept may not be worthwhile.

---

[1] Ada is a registered trademark of the U.S. Government for the Ada Programming Language defined in ANSI/MIL-STD-1815, commonly referred to as the Language Reference Manual, LRM. At the time of preparation of this document, revision A dated 22 Jan 83 was effective.

In particular, we do not anticipate that Software BIT will useful when applied to:

- Software systems for which the execution control is explicitly synchronized by an application-specific executive which accepts responsibility for sequencing and controlling inter-process interactions (as opposed to a modular decomposition into conceptually "independent" processes a la the Ada Task concept)

- Specific failure mechanisms for which a direct, explicit test is available. (As a general rule that it is much more effective to test and respond directly to each anticipated failure mechanism than to use a generic test which might be more difficult to interpret.)

Finally, it is worth addressing the resource issue in this summary. As in most things, the effectiveness of Software BIT will be proportional to the resources allocated to the task. Within the context of real-time systems of the sort required by SDIO, there is a natural tendency to assess a severe penalty during the design phase for anything that consumes either processing or storage resources without directly contributing to functionality. Nevertheless, industry experience with highly reliable systems (especially within the context of communications switching systems) has been that it is often desireable to devote as many resources to the diagnostic and maintenance tasks as are devoted to providing functionality; but these resources are not wasted. They provide the specific functional capability for highly reliable system operation.

Within the DoD community, there is a widespread recognition that it is necessary to preserve expansion margin within a system design to permit downstream evolution; but, there is also a reluctance to devote resources to measurement and instrumentation. We believe that this practice should be modified to permit use of spare processing and memory capacity for on-line instrumentation during early deployment, especially if it is coupled with algorithm learning. Thus, while system evolution might require a reduction in instrumentation as the system matures, initial operational versions would be as widely instrumented as possible to permit data collection about the true

operational environment. (For example, use of spare memory to capture operator input buffers for a command and control system provides very useful post-mortem information at minimal cost provided the memory is available. As system evolution requires some of the memory, buffer size can be reduced.)

Preface

This effort is a result of a proposal made to SDIO under the DoD Fiscal Year 1986 SBIR solicitation. Work began in February 1987 on a contract administered through ONR. The startup delay inevitably caused changes in emphasis: both as a result of additional Atlantis experience and as a result of changes in emphasis within the SDIO program. Nevertheless, the research reported herein is consistent with the concept initially defined in our proposal.

Several people influenced the direction and presentation of the research. Our contract monitor, Dr. Neil Gerr of ONR, was invariably helpful and innovative in attempting to keep us oriented in the right direction. Mr. Leonard Caveny of SDIO was most helpful on the several occasions we needed to interface with his organization. Finally, a special note of acknowledgement is due to Mr. Doyce Satterfield of the Army's Strategic Defense Command, whose courtesy and interest have had a significant influence on our understanding of how these results might relate to an important part of the SDIO program. We sincerely regret that we were unable to establish contact with his organization earlier in the period of performance.

Table of Contents

1. Introd' ction

This report presents the results of an investigation of the application of the concept of Software Built-in Test (BIT) for use in support of the objectives of the Strategic Defense Initiative Organization (SDIO). The investigation was sponsored by the Department of Defense (DoD) Small Business Innovative Research (SBIR) Program and was administered by the Office of Naval Research (ONR) under Contract No. N00014-87-C-0134. The results reported herein are for Phase I, feasibility evaluation, of the SBIR effort. The work was accomplished during the six-month period February-July 1987.

1.1 RESEARCH OBJECTIVES AND CONTRACT ACTIVITY

The technical objectives from the Phase I proposal are to:

- Determine whether it is possible to define a software analog to the hardware concept of Built-in Test (BIT)
- Identify potential applications within the SDI schedule constraints
- Design an experiment to demonstrate the utility of Software BIT for detecting off-nominal performance in the SDI context.

The proposed technical approach identified three tasks to be accomplished:

- Perform a literature review and formulate the problem specifics
- Develop candidates for Software BIT
- Develop a Preliminary Design for a Phase II Experiment.

During the literature review, it became clear that the concept had both merit and commercial precedent within the telecommunications industry. (See Section

2.2.4.) Additionally, we were able to define four generic categories of Software BIT that appear to include all of the numerous examples of both computer-specific and application-specific Software BIT. Moreover, the BIT can be implemented using a software tool which places minimal demands on the software designers and implementors.

We achieved only limited success in identifying appropriate SDI applications as candidates for the proof-of-concept experiment. Initial efforts to coordinate our efforts with other SDI applications, with which Atlantis personnel were familiar, did not identify suitable applications because the research objectives of those efforts had been changed to concentrate on different priorities during the time between submission of our proposal and initiation of contract activity. Late in the contract period of performance, we were able to identify the Advanced Research Center (ARC) of the Army's Strategic Defense Command as a likely target for a proof-of-concept experiment, but our data collection visit of 23 July was too late to attempt an experiment design then.

As a result, we concentrated instead on developing the concept of a BIT generation tool and were not totally successful in achieving all of the research objectives.

1.2 REPORT ORGANIZATION

The report is organized into four chapters and an appendix. Chapter 1 is an introduction and describes both the research objectives and report organization. Chapter 2 contains a description of the concept of Software BIT, specific examples of computer-specific and application-specific BIT, and the four generic categories of BIT we have identified. Section 2.3 also contains a description of selected characteristics of real-time software that are important in determining the applicability of Software BIT to a particular application.

Chapter 3 describes many of our Phase I research results as they affect the implementation of Software BIT. In particular, Section 3.1 describes three different strategies for responding to a BIT alarm. Section 3.2 provides an

illustration of the concept of "training the system" to adjust thresholds and alarms based on system experience gained during testing and non-crisis operation. Section 3.3 identifies two aspects of the SDI application that will influence any practical implementation of Software BIT. Section 3.4 presents implementation considerations and includes a description of a generic software tool for implementing Software BIT. Section 3.5 summarizes a brief investigation of the role that the choice of software development language might play in the application of Software BIT to practical systems.

Chapter 4 presents conclusions and recommendations for further effort. The appendix contains an evaluation of the role that the Ada exception construct can play as an implementation of Software BIT.

2. Software Built-In Test (BIT)

2.1 BACKGROUND AND RATIONALE

2.1.1 Background

Software is an important component of all of the current architectures for accomplishing the SDI mission. As a result, correct software processing is essential to SDI mission effectiveness. Software correctness will be established through:

- Design process
- Test process
- Laboratory and feasibility demonstrations
- Pre-crisis operational experience.

The credibility of software correctness will depend on the effectiveness of the management process and its documentation.

Software failures are substantially different than their hardware analogues; software doesn't break or wear out. Instead, software fails to operate correctly when:

- Design process is flawed -- poor or incorrect specifications, inadequate design partitioning, and the like.
- Operation is attempted in a "new" and/or "untested" environment.

Managing the design process can significantly reduce errors introduced through the design, but such activity is beyond the scope of this effort.

This effort focuses on developing the concept of Software BIT for SDI software. The concept involves measuring software performance and comparing it to (1) performance limits derived from design values for expected performance or (2) performance limits adjusted during test and pre-crisis operation.

Atlantis believes that the concept of Built-in Test (BIT) can be expanded to include software components of BIT. The primary purpose of the software BIT would be to monitor the operational software to determine when the software is executing outside expected windows of performance. In the event that the BIT triggers, one of three things has happened:

(1) the external environment has changed in an unexpected way

(2) the software has failed (or a failure is imminent)

(3) the hardware has begun to fail.

Therefore, there is a significant probability that a software error has occurred or is about to occur. The concept would be particularly attractive during test and pre-crisis operation because it would allow timely analysis and software correction. However, there appears to be significant potential payoff during the crisis operation mode if the software BIT could be incorporated as a factor into the battle management scheme.

2.1.2 Predicting Software Failures

We believe that a primary cause of software failure in properly designed and tested software is an attempt to operate the software in an environment other than the "Design Environment". Figure 1 shows the ideal relationships among the "Design Environment", the "Operational Environment" and the "Test Environment". In this case, the Operational Environment would be a subset of the Design Environment. In this way, the system is guaranteed to operate only in an environment for which it was designed. Additionally, the Test Environment would be totally contained within the Design Environment and would attempt to cover as much of the Operational Environment as resources

permit. Note that it is conceivable that the Test Environment would grow with time, as more tests are performed. (It is highly probable that the SDI Test Environment will grow continuously for a long period as additional laboratories are built and new operational strategies evolve.)

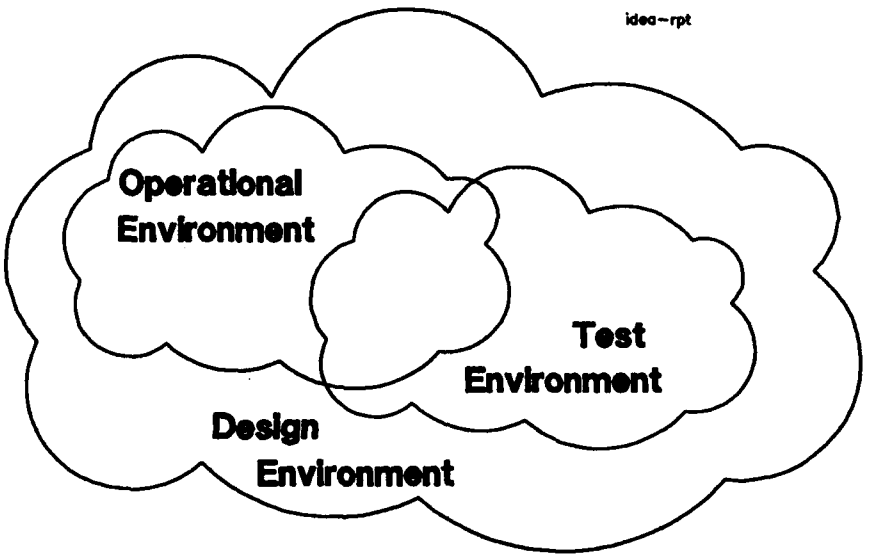

Figure 1. Ideal Environment

Figure 2 shows the more normal relationships among these environments. There is usually significant overlap between the Design and Test Environments, some overlap between the Design and Operational Environments, and some overlap between the Test and Operational Environments. As long as the system is operating within the Design Environment, there should be little risk of failure. Within the region for which it was tested, the risk is also low. The risk is highest in that region where it is asked to operate outside the Design Environment and where it has not been tested.

The potential value of Software BIT lies in providing a practical way to determine that the software is operating outside the Design Environment. It also seems highly desirable to indicate that it is outside the Test Environment, as that will reduce the sensitivity to design errors.

norm-rpt

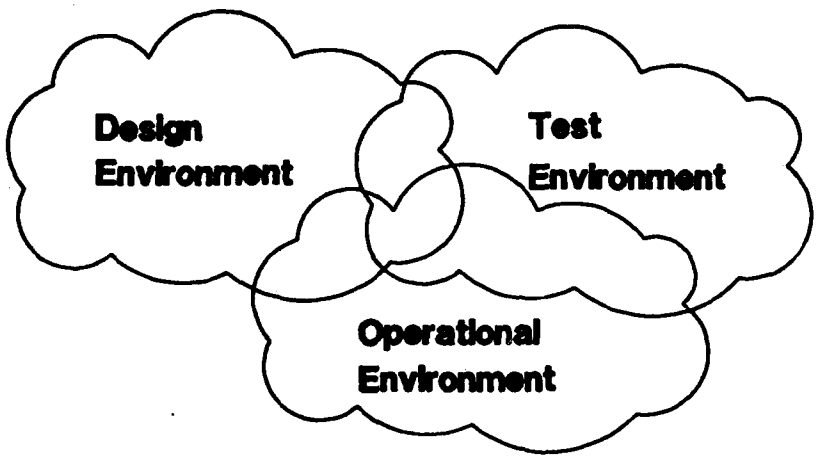

Figure 2. Normal Environment

2.2 THE BIT CONCEPT

Hardware designers have been incorporating BIT into sophisticated equipment for some time. By measuring certain quantities at selected test points within the hardware and comparing the test point values with pre-selected thresholds, failures or off-nominal performance can be detected. Once the "failure" is detected, an operator is notified, or the equipment takes "appropriate" action to protect itself and even, at times, the operator. Proposals for "smart" BIT usually include an attempt to correlate these measurements with measurements of the external environment (e.g., temperature) to improve BIT effectiveness and reduce false alarms.

2.2.1 Software BIT

The purpose of software BIT is to use the operating software to determine whether the current operational characteristics of the software are consistent with expectations and measurements for the original design environment, prior operational or test experience, and/or specific design criteria established during the BIT design process. There are basically two kinds of parameters that can be sensed:

- Computer-specific

- Application-specific.

Computer-specific features are sensed by the operating system, while application-specific parameters must be built into the applications software. (Note that not all operating system parameters of interest will be available from all operating systems.)

2.2.2 Role of the Operating System

Modern software practice uses the concept of an operating system to isolate the applications program from the external environment. Although many currently fielded DoD systems do not have a formal operating system, it will be useful to distinguish those features of the software which perform operating system functions from the application functions for purposes of this discussion.

The operating system separates the application software from the external system environment and handles certain scheduling, resource allocation (i.e., I/O), and interrupt handling functions. It uses queues, linked lists, device drivers, and a number of other software tools to perform its function. It may also use some special hardware features of the computer to guarantee success; e.g., real-time clocks and resettable counters.

Although the operating system uses these tools to perform its function, most operating systems don't provide convenient user access to these parameters. Significant effort may be needed to sense certain performance features such as interrupt response time, page faulting through the memory management system, and other "application program transparent" features. (This is particularly true for more modern processor/ system designs. For example, much current generation microprocessor software uses "software interrupts" to provoke context switching making this function virtually identical to I/O processing and very difficult to effectively predict.) Nevertheless, Atlantis believes that there is high potential payoff for developing relatively simple Software BIT parameters within the operating system.

Because the operating system is the link (at least conceptually) between the applications software and the environment, it is the level at which the environment must be monitored. More importantly, it is the place at which significant differences between the real world and the design environment can be measured.

2.2.3 Application-specific Tests

A number of application-specific parameters are important in assessing whether the system is being operated within the envelope of the Design Environment. Unfortunately, these parameters are peculiar to specific applications; we are not in a position to provide examples particularly relevant to SDI. Nevertheless, some functionally relevant examples are appropriate.

E-3 NAVIGATION EXAMPLE

One of the tools frequently used in modern mathematics as a basis for combining successive measurements to obtain "optimal" estimates is the Kalman Filter. The Kalman algorithm is based on an a priori statistical description of the environment. Whenever the environment conforms to this description, the filter performs "optimally". Various techniques have been developed to de-sensitize the Kalman Filter and let it "adapt", but the algorithm is fundamentally limited by the necessity to operate within the Design Environment. If the Operational Environment differs from the Design Environment, some performance degradation may occur. The divergence between the Operational and Design Environments has been the cause of numerous Kalman Filter re-designs and "software re-works". Nevertheless, it is possible to check on the "reasonableness" of individual measurements and compute a "goodness of fit" parameter (the literature is full of numerous examples of using such parameters to drive adaptation algorithms) which could be used to detect the divergence. Such a parameter would be a candidate for an application-specific software BIT.

As a concrete example, the E-3 (Air Force AWACS Aircraft) Navigational Computer System (NCS) uses an integrating Kalman Filter to combine data from the Omega radionavigation system, the navigator, and a doppler radar to aid the on-board dual inertial navigation sets. When the system was designed in the early 1970s and flight tested in 1975-1976, it was believed that the Omega radionavigation system would achieve the worldwide 1-2 nm,rms position-fix accuracy experienced in the test region and advertised by the system developer. In the early 1980s, it became apparent that the overall system accuracy was more like 4-8 nm (95%). As a result, the E-3 has experienced difficulty achieving specified navigation accuracy in certain geographic regions. It is expected that a new version of the Kalman Filter, tuned to the less accurate Omega radionavigation system, will be fielded in 1986. In this instance, it is clear that the difference between the Design and Operational Environments led to substantial degradation in system performance. (Note that the E-3 remains mission capable because the design permits navigator intervention, but the crew workload is significantly higher than anticipated!)

In validating the software for the next release of the E-3 NCS software, a data collection package was built to monitor the Kalman Filter measurement residuals, and data from the old and new software were compared in various geographic regions. As a result of the data collection effort, it was decided to implement the new version; but the important point is that the data collection package was monitoring an application-specific software indicator of system performance to determine whether it was outside the design environment. (I.e., the package was using Software BIT.)

A-7 WEAPON RELEASE EXAMPLE

Weapons delivery software is frequently sensitive to the geometry between target and launch platform, with some geometric aspects highly preferable to others. As a result, system performance will depend on whether the system was/was not in proper aspect at the weapon release point. These geometric parameters, which drive the system error budget, are good candidates for application-specific software BIT.

In conjunction with the installation of a new hardware weapon system, the software on the A-7 (Navy Attack Aircraft) was significantly altered in the mid 1980s to integrate the new system. During OPEVAL, it was discovered that the bomb-navigation system was no longer capable of scoring accurate hits with gravity bombs when employed by operational pilots. This discrepancy provoked considerable consternation until it was determined that the real problem was that the pilots were now flying significantly different mission profiles in order to accommodate improvements in threat anti-aircraft capabilities. This change in operational employment took the weapon system outside the Design Environment for the gravity bomb algorithms. As a result, considerable effort was applied to the task of algorithm redesign.

In this case, simply monitoring the height-above-target at which the weapons were released would have provided a useful figure for anticipating weapons delivery accuracy. Since this data value is an input to the ballistic calculations, it is readily available. Hence, height-above-target is a useful candidate for Software BIT in this application. It could have been used to alert the operator that the system was operating outside the design envelope for accurate weapon delivery.

2.2.4 Commercial Examples

As part of our Phase I effort, Atlantis performed a literature search to determine if the Software BIT concept has been implemented in other areas. We concentrated on both the electric power and telephone communications industries as potential candidates in addition to the DoD literature available through the Defense Technical Information Center (DTIC). It appears that there is ample evidence to infer that a system similar to the Software BIT concept is currently used within at least one telephone switching system to determine when potential problems exist, trigger additional maintenance diagnostic testing, and influence resource allocations. (See References 1-3.)

2.3 SELECTED CHARACTERISTICS OF REAL-TIME SOFTWARE

There are numerous ways to characterize real-time software. For our purposes, two features are appropriate: the method by which individual software components are scheduled, and the execution phases during operation. For simplicity, we refer to individual software components capable of distinguishable operation as tasks (clearly intending the Ada context definition but accepting other interpretations) and divide execution into three phases. These characteristics are described below only to form a context for discussion of Software BIT and for clarification.

2.3.1 Phases of Real-time Software

Each task involves three phases: initialization, execution and termination. During initialization, the task must create its own internal logical environment, initialize variables and buffers, etc. The execution phase represents the major focus of the system design. (Our classification scheme is based on the method by which the task is controlled during the execution phase.) Finally, it is necessary for the task to terminate properly. Fig. 3 represents these phases schematically.

Phase changes are scheduled by the task but triggered by external processes--usually the operating system. Tasks may execute once, indefinitely, or a fixed number of times.

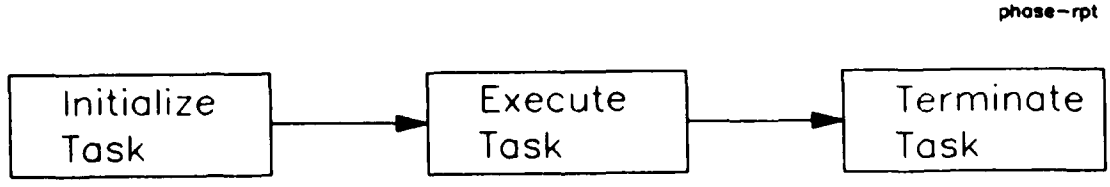

Figure 3. Phases of Real-time Tasks

2.3.2 Categories of Real-Time Software

From the perspective of task control, three different strategies can be applied during the execution phase: synchronous, explicit, and asynchronous control. Each has benefits as well as drawbacks. Software categories are defined in terms of the method in which tasks are controlled during execution phase.

Please note that we deliberately exclude any operating system software from these discussions.

SYNCHRONOUS CONTROL

Figure 4 depicts the activation schedule for a group of synchronously controlled tasks. In this approach, time is divided into uniform intervals (frequently called frames) and all tasks are activated in each interval. The amount of execution time allocated to each task may depend on the task itself and is usually not controlled explicitly (i.e., each task executes to completion under its own control). Time slices are not necessarily equal, but the task invocation sequence is fixed (i.e., task 1 always executes before tasks 2, 3, ...).

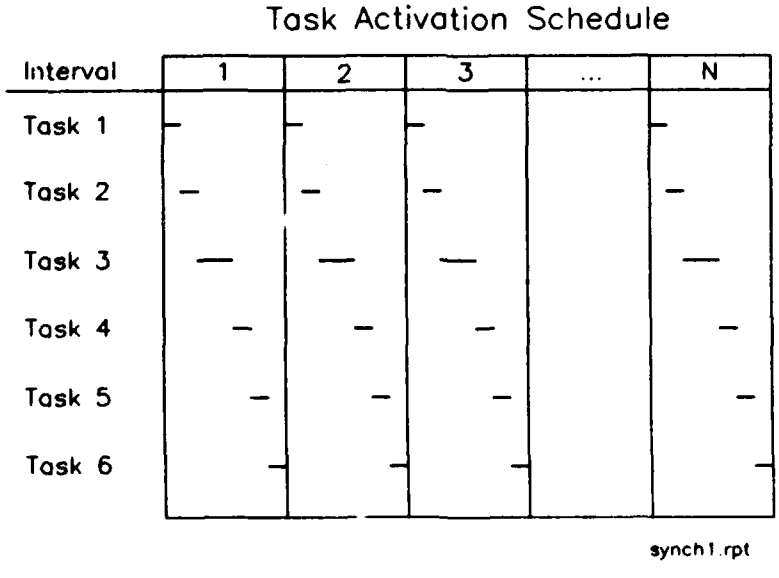

Figure 4. Task Activation Schedule for Synchronous Control

This form of control involves very low task overhead. Since the execution sequence is fixed, synchronization is not difficult. Data dependencies are predictable, but the inter-task interactions may be complex. Maintenance of this type of software can frequently be very difficult as even small changes to a single task may involve changes to most other tasks in order to accommodate execution-time and data dependency constraints.

EXPLICIT CONTROL

A significant increase in flexibility is achieved by designing an explicit control strategy that schedules tasks as they must execute rather than uniformly, as above. In this approach, uneven intervals are allowed. Fig. 5 depicts such an activation schedule.

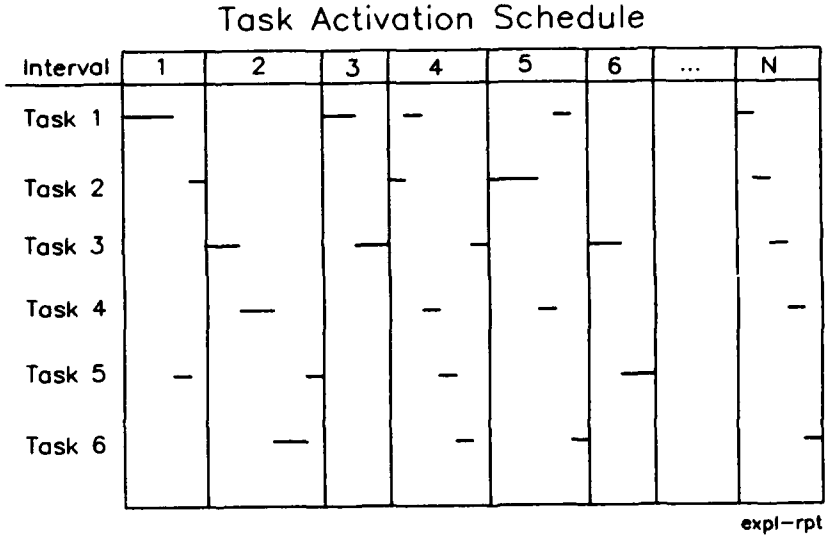

Figure 5. Task Activation Schedule for Explicit Control

Normally, this type of control strategy increases throughput at the expense of increased design effort required to manage the task timing and interrelationships, data dependencies, error detection and recovery, and task initialization and termination. It tends to reduce task overhead somewhat, as each task assumes that it is ready to run when invoked. Maintenance of this type of software can be very difficult.

ASYNCHRONOUS CONTROL

Asynchronous control involves scheduling tasks to be activated as required during program execution using explicit calls to an operating system. In this manner, tasks schedule themselves. Normally, this scheduling process involves assignment of priorities to each task and some form of priority arbitration. As a result, task overhead is high for this control strategy. Tasks must maintain their own control of time and usually spend much longer in such overhead tasks as regaining synchronization, scheduling the next event and checking error conditions. If precise timing is important, it is usually up to the task itself to achieve it.

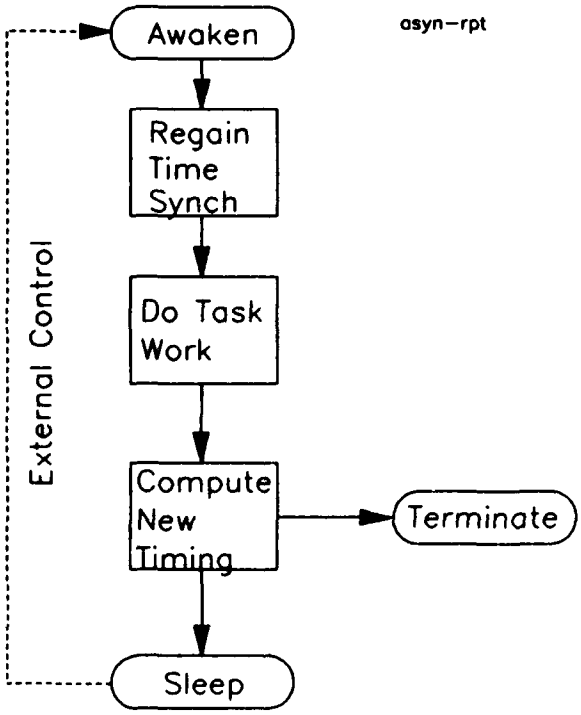

Figure 6. Schematic Representation of Asynchronous Control

Figure 6 depicts the asynchronous control strategy. Each task appears to operate independently of all other tasks. The task terminates voluntarily by either a "Sleep" or a "Terminate" call to the operating system. The relationship of one task to another must be controlled explicitly and frequently involves flags and semaphores.

Software that is asynchronously controlled is frequently described as "unpredictable", as the processor time required for execution frequently depends on the data encountered. Nevertheless, such systems can be quite efficient in their use of computer resources, and "background" tasks can be scheduled to take advantage of resources that would be wasted otherwise. Most multi-processor systems are controlled asynchronously.

2.4 GENERIC CATEGORIES

During the course of this effort, it became clear that it would be necessary to develop an automated tool to help implement Software BIT within the anticipated SDI BM environment. As a result, Atlantis analyzed the types, categories and examples of Software BIT identified to date and defined four generic categories of BIT. These categories are:

- event counters
- rate samplers
- range checkers, and
- data samplers.

Event counters are triggered by specific events and report the occurrence of those events; e.g., a counter used to indicate receipt of a garbled message or the occurrence of a "warm" boot. Rate samplers are used to determine the rate at which events occur by sampling event counters over known intervals (which may be regular or asynchronous depending on the application) to determine the

average frequency of occurrence of events; e.g., the rate at which garbled messages are received (which might be an indication of channel deterioration). Both classes of BIT can be significant with and without design range limits.

Range checkers are used to compare local variables with anticipated performance design limits and can be implemented either as maximum (minimum) value or interval checkers; e.g., the temperature gauge on an internal combustion engine is normally marked to indicate the range of acceptable performance. The BIT indication is that the variable is out-of-tolerance. Data samplers provide a means of recording current values of local variables to provide post-mortem clues to performance and are generally not associated with hard design limits. Data samplers are the least specific type of BIT.

Each of these classes is discussed in more detail in subsequent sections.

2.4.1 Event Counters

Event counters detect and report the occurrence of rare events and/or infrequently executed portions of the software. Specific examples are restart and retry counters, process activation counters, watchdog timer elapsed, and error handler invocations.

Event counters instrument portions of the software that have been designed to respond to abnormal environmental or operational situations. In many single computer systems, they would be reported to a maintenance log (possibly a hard-copy terminal) to be available to the service technician/ system engineer as data to be used in evaluating performance or trouble shooting system faults. Within the SDI environment, it will obviously be necessary to report the occurrence of the event to some external agent (implying some form of message generation).

2.4.2 Rate Samplers

More often than not, the true indication of a potential system problem is related to the frequency of occurrence of events rather than to the events

themselves. For example, most communications systems are designed to tolerate noisy communications channels. The channel is characterized in terms of a bit-error-rate, and protocols are designed which compensate for data loss at or near the maximum channel bit-error-rate. (A relatively error-free channel with a bit-error-rate of 10^{-7} will average several faults per hour if tens of 100-character messages are transmitted each minute.) As a result, the total number of errors is not a good indication of performance. What is required is to determine whether the error-rate exceeds allowable tolerances.

Rate samplers accomplish this task, but require a more elaborate implementation than any of the other forms of generic Software BIT. By periodically sampling event counters, rates are determined and compared to design thresholds. Implementing this within a multi-tasking environment involves scheduling a sampling task for execution on a periodic basis. The sampling task can determine from the system clock the amount of time which has elapsed between successive invocations and thus determine the event rate.

2.4.3 Range Checkers

Range checkers compare an identified variable with a predetermined range of acceptable values to determine whether an alarm should be set. Range checks can be applied both to computer-specific variables, such as queue and stack lengths, and to application-specific variables, such as engine operating temperature.[2] We anticipate that range checkers will be the most common form of BIT.

Although it is frequently possible to select useful variables to subject to range checking, it is often difficult to set an appropriate range. Therefore, it is desireable to provide a mechanism for adapting the thresholds during test and non-crisis operation in order to gain operational experience. This leads to the

---

[2] Engine operating temperature is clearly hardware BIT rather than Software BIT oriented. Nevertheless, we have found it a more useful example than any software concept because there is no need to explain its significance. Further, most people recognize it is a two-sided test, since engines that operate "too cool" are often just as "out-of-tolerance" as engines that run "too hot".

concept of training the algorithm, discussed in more detail in section 3.2. By permitting a training mode, it is possible to adjust the threshold tolerance to provide meaningful, experience-based operating limits.

2.4.4 Data Samplers

In order to accommodate the wide range of potential candidates for application-specific Software BIT, it is necessary to define a special purpose tool which permits capturing specific information on particular variables. No threshold testing is involved. Rather, the data value is reported each time the BIT is executed.

Note that no training thresholds will be used for this type of BIT. However, it might be advantageous to use the "New Threshold Set" message as the mechanism for reporting data values. Data samplers are not a substitute for software instrumentation.

3. Implementation Considerations

3.1 PARTITIONING THE RESPONSE TO A TRIGGER

Three different strategies exist for reporting BIT alarms:

- Direct action to change the control flow of the program in response to the out-of-tolerance alarm. (An example of this is the Ada concept of EXCEPTION and EXCEPTION HANDLING.)

- Direct action to initiate alarm notification without significant interruption to the control flow of the program. (Execute one path of an IF-THEN construct to "send a message" to the maintenance function but resume program execution at the next operation after the BIT test instruction.)

- Indirect action to post an (or one of several) out-of-tolerance condition warning for subsequent processing by a BIT response task which must search through the BIT alarms capable of posting the warning to determine which alarms are active. (The out-of-tolerance condition acts as an alarm annunciator for the maintenance task causing it to undertake a complete review of system status.)

Each is worth elaboration within the context of real-time systems.

Direct action to change the control flow of the program is appropriate when either the alarm is sufficiently serious or the penalty for aborting the current process is low enough. Examples include: a ground proximity detector which might cause a terrain-following radar system to stop processing detailed radar returns and issue a "pull-up" directive to the flight control processor; stack-overflow detection; or, a telephone switch center which might simply disconnect the call in process in order to initiate maintenance which puts the node (or trunk set) out-of-service. Direct action to change the control flow of the program normally creates a technical roadblock to clearing the problem and

resuming execution where the program aborted. (Note: the latter problem is often cited as a criticism of the Ada EXCEPTION concept, since the EXCEPTION HANDLER cannot resume program execution within the block in which the EXCEPTION was RAISED. This is less a valid criticism of Ada than it is an indictment of sloppy engineering practice; however, the problem does limit the use of EXCEPTIONS more than the early Ada advocates and teachers thought.)

Direct action to initiate an alarm notification without significant control flow interference is a less drastic step than the first. It is appropriate whenever the potential for immediate damage is not high enough to warrant immediate intervention and the current process can continue. Since the notification message can be tailored (and/or readdressed) depending on the actual BIT indication, this is a very flexible form of response. It also represents a relatively high overhead response.

Indirect action is the lowest overhead response. In general, it relies on another process to determine the severity of the problem and to initiate a response. Action continues after the BIT alarm is raised. Thus, the technique can be applied in software for which very predictable execution scenarios are required. In general, it implies that the "out-of-tolerance" condition can also be detected by another process so that at least some of the BIT must be accessible in either global or shared memory.

Note that the distinction between the last two categories is necessarily vague. The concept is one of taking explicit "corrective action" versus raising a "hey look at me" type flag. Neither is as direct as the first category (which happens to be the one defined in Ada and mostly ignored/overridden in practice).

The notion of local versus global storage is important, but it is not yet well developed. One of the serious flaws in the original concept is the requirement to share data among processes. This conflicts with the software engineering principal of "information hiding" and can cause trouble. By using direct action without intervention in the control flow, the BIT can avoid sharing detailed

information. Thus, it is an important refinement of the concept resulting from our Phase I investigation.

3.2 TRAINING THE SYSTEM

One of the concepts that was described briefly in the proposal could be called "training the system". It involved using development and integration testing as an opportunity for determining what alarm thresholds should be built into the operational software. Moreover, it is possible to use controlled operational testing and exercises (when the user knows that he is not in "crisis operations mode") to help determine acceptable limits, which can then be set into the software for use during "crisis operations mode". (Recall that the principle is to detect when the software is operating in a "new" environment so as to preclude operation under circumstances in which there is neither existing operational experience nor design rationale for expecting correct performance from the system.) Trainability is a direct result of maintaining simplicity in BIT design.

An example will illustrate the point. Suppose that the indicator of interest is one which should remain below a certain threshold. The following pseudo-code representation of the trainability algorithm is appropriate:

```
IF (Indicator > Alarm_Threshold) THEN {
        IF Train_Mode  THEN [ Reset Alarm_Threshold to Indicator;
                              Post New_Threshold_Set_Indicator]
                       ELSE [ Process Alarm Indicator ] }
```

This is a self-calibrating threshold whenever the program is in Train_Mode. Moreover, the software can be balanced to ensure approximately equal path execution whenever an alarm exists. It should be included in the operational software, not deleted prior to delivery. Thus, it is possible to use it in real-world user exercises.

Note that it is necessary to notify someone if a threshold is trained, especially if particular thresholds are stored in "local memory" so as to "hide the

information". In that case, a special storage task may also be required in order to "permanently save" the new threshold. Our preference is to incorporate this data save function into the normal, orderly shutdown of the task.

3.3 SDI IMPLEMENTATION CONSIDERATIONS

3.3.1 Fault Tolerance

Although few relevent decisions have been made about the SDI implementation, it is certain that fault tolerance will be an important design requirement. Exactly what form that will take is not clear and may not be known for some time. However, it is clear that fault-tolerant systems invoke some software design features not always present in other types of real-time systems. These features are important considerations in designing Software BIT. Two such features are:

- Automatic Retry[3]
- Automatic Restart

Both design features attempt to compensate for the effect of errors detected during system operation. (Whenever the action is pre-programmed, it is termed "automatic". We assume that the timeliness involved in SDI applications will lead to substantially automatic system responses.) The normal interpretation of retry is that the system makes a second attempt at an operation which failed and continues if the operation is successful. (This is particularly useful in correcting "transient" faults.) Restart involves reinitializing some part (possibly all) of the software to a known good[4] state. (The terms "cold", "warm" and "hot" restart are frequently applied to distinguish among attempts to preserve none, some, or all of the tasks in process at the time of the restart.)

---

[3] For terminology, see Ref 6.

[4] We distinguish between "good" and "error-free", since it is nearly impossible to assert that the system is error-free unless a cold restart is attempted.

Event counters that monitor the number of retry and restart actions for each separately identified major task are excellent candidates for Software BIT. Restart counters can be incorporated into the task initiation process, if either warm or cold restarts are used, without severe execution penalty. Retry counters must be included in the fault response logic and may impose a measurable execution penalty. In addition, Atlantis anticipates that rate information may be useful for many types of tasks. (Rate measurements are obtained by sampling the event counters at known intervals and may be implemented as background tasks. Wallace, Ref. 7, reports using short-term error counts and thresholds to detect failures and long-term counts to help detect erratic, transient behavior.)

3.3.2 Continuous Operations

Designing systems for continuous operation over extended periods imposes additional design requirements. In particular, it requires that the system be capable of adapting to changes in both hardware and software. Hardware components will be added, subtracted, repaired and redesigned during the useful life of the system. It would seem naive to expect that the software will not change also.

Two features of continuous operations have significant implications for Software BIT. First, any implementation of a continuously operating system contains design features that will be candidates for Software BIT. For example, the program loaders, hardware configuration data loaders, and configuration data managers should all include event counters which are BIT candidates. Additional candidates will depend upon both the design and the degree to which individual candidates are monitored directly by the system as trouble indicators.

The second significant implication is that any realization of Software BIT must reflect continuous operation. In particular, the test points, limits and responses must be linked through a dynamic Software BIT manager that is capable of: allocating new test points, eliminating old test points and results from the database, changing severity codes and responses, and monitoring its own

operations and status. These conclusions follow directly from the implicit requirement for evolution within any continuously operating system.

3.4 IMPLEMENTATION

Implementation of Software BIT involves a number of different software components:

- instrumentation software -- additions to the operational software that perform the test
- reporting software -- additions to the operational software to report the test software external to the module being tested
- response software -- additions to the system software which respond to the BIT indication and re-schedule or re-allocate resources and/or announce events to human operators.

Instrumentation software is described in section 3.4.1 below. For the general case, reporting the occurrence of a BIT event requires interaction with the on-line operating system and is described in section 3.4.2. Response software is beyond the scope of this report; moreover, it is extremely application dependent and may vary considerably across various SDI applications.

Selection of the variables to instrument, assignment of thresholds and severity indicators, and interpretation of interim results is very labor-intensive and will require involvement by the software development professional at the module level. Implementation of the reporting mechanism is not module dependent; rather, it is project dependent. As such, we believe that it will most likely involve project tool-builders and systems engineers. Moreover, it is influenced more by the architecture of the system and the selection of the operating system than by the specific application. Therefore, Atlantis believes that an automated tool may represent the best project solution, both from the technical implementation perspective and from the project standard perspective. These topics are described in sections 3.4.3 and 3.4.4, respectively.

3.4.1 Instrumenting the Software

Adding any of the four generic types of Software BIT involves selecting the variable or path to be monitored and setting specific thresholds (i.e., the range limits for type 3 BIT and sampling period for type 2 BIT). This is a very mechanical process that can easily be automated using compiler directives (e.g., the Pragma concept in Ada) in almost any language.

For example, implementing an event counter is a matter of inserting instructions which "increment a counter and report the event" into the instruction stream of the appropriate path. For strongly typed languages, the event counter must be "declared" within the appropriate block; it may be necessary to constrain the variable to be "static" in order to ensure relative permanence within the task environment.

Reporting the occurrence of the event is a matter of formatting an appropriate message to the maintenance manager. There will be different implementations of the reporting mechanism depending on the control-flow interaction selected for the project. (See section 3.1.) Nevertheless, the process is one of reporting the time and type of event together with an indication of where it occurred.

3.4.2 Interacting with the Operating System

The Software BIT implementation must interact with the executing real-time operating system in at least three ways:

- Identification of tasks and processes
- Message routing and passing
- Implementation of rate sampling.

In addition, it appears highly desireable that the implementation interact with a dynamic database of values and thresholds which may also involve the operating system.

During the initialization phase, it will be necessary for each BIT implementation to obtain the project-unique identifier for the task (or process) it is monitoring. Although it is conceivable that processor and task identification might be assigned during software development, it seems much more likely that they will be assigned dynamically during system operation and that it will be necessary to obtain them at run-time from the operating system. These identifiers are used in conjunction with the reporting software to identify the source of BIT alarm and threshold adjustments.

Once a BIT indicator is triggered, it must be reported. This requires sending a message from the operational task to the maintenance manager informing it about the event. Different strategies can be used for formatting, sending and receipting the message; Atlantis evaluated a number of them during the Phase I effort. Our preference is for a simple process modelled after the old control log concept in which a teletype at the operator's console was used to record significant events for review by the operator on an as-required basis. In this approach, a logical message file is maintained by the operating system and transferred to the maintenance manager as circumstances warrant. (The system design can provide for periodic or event-driven message transfer. A communications task can be added to each processor if the operating system does not provide this feature automatically.) Messages are time-stamped when recorded in the logical message file. This time-stamp is used by the maintenance manager to sequence events for analysis and permits maximum flexibility in message routing. We anticipate that BIT messages will be assigned a service precedence and transmitted on a resource-availability basis. (Table 3-1 depicts the categories and likely contents of Software BIT messages. Note that the distinction between internal and external distribution refers to whether or not the BIT message leaves the local processor to interact with the on-line database.)

Table 3-1. BIT Message Categories

Category	BIT Type	Distribution	Message Content
Event	All	External	Task_ID, Event_time, BIT_ID, Event_type, Remark
Threshold	3,4	External	Task_ID, Event_time, BIT_ID, New threshold value
Data Request	3	External	Task_ID, BIT_ID
BIT Coordination	2	Internal	BIT_ID, Additional implementation details (TBD)

Implementing the rate sampler also involves the operating system as an independent sampling task is required. In the general application, a sampling task will be scheduled to sample each event counter at the appropriate rate on a non-interference basis. However, we anticipate that most applications will prefer to have a single sampling task implemented on each processor rather than one sampler task per event. Neither approach is conceptually difficult, but the single-task-per-processor approach may involve a more elaborate sampling task which reduces the load on the operating system.

3.4.3 The Generic Tool

In the context of the distributed, fault-tolerant, continuously operating architecture required to implement the battle management task in SDIO, it is evident that implementation of Software BIT will require tools in at least three different categories:

- Preprocessor Tools
- On-line (possibly distributed) BIT Data Manager

- On-line BIT Annunciator

Each category is described briefly below.

Preprocessor tools operate in the development host computer (for Ada this would be the APSE) either as a preprocessor for the compiler or as an adjunct to it in order to insert the necessary process-unique identification and routing information into the source code. The programmer is expected to realize that a "Type 3 BIT is required to monitor local variable XXXX" at a particular point in the program, and the preprocessor must implement that concept. A conceptual representation of this process is provided in Fig. 7.

> NOTE: This is largely a configuration management and data communications routing task. The issue here is to ensure that the overall BIT report is never confused about where an error or indication comes from, what it means, etc. It is too much to expect that the individual programmer should be tasked with resolving the BIT implementation issues.

The on-line data manager receives BIT reports from the operating tasks, updates the BIT database, and prepares (and possibly posts) operational status reports. This category is relatively straightforward but will be project unique (i.e., the SDI Battle Manager may have a different on-line database than the tracking and pointing systems, etc.).

The on-line annunciator processes the information from the software BIT to provide inputs to the Battle Manager. Atlantis can offer little insight into what this portion will be for the SDI application. In a simulation test-bed, we envision that a console display capable of presenting status information to the facility manager might be appropriate. It is clear that the form and format will vary depending upon whether or not a human operator will be asked to interpret the information.

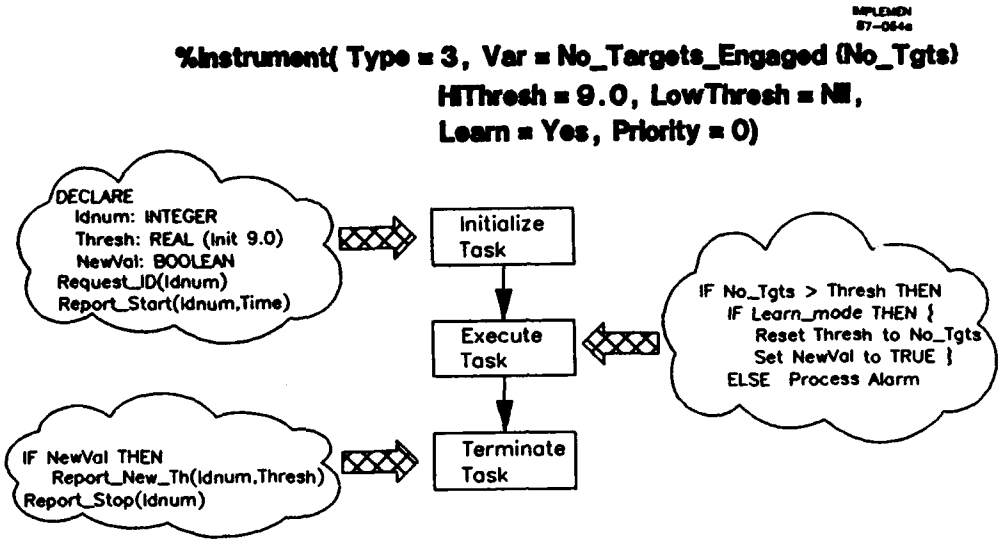

Figure 7. Thematic Representation of Pre-processor Tool

This is the province of the evolving technology: the decision aids, artificial intelligence, etc. It is real and certainly important to the overall application of Software BIT to SDIO applications, but it may be premature to consider it now.

3.4.4 Making Software BIT Happen

During the course of our research, Atlantis personnel discussed our concept with numerous software and system professionals. Most were quick to point out that they had some experience with attempting to monitor system performance using tests similar to Softare BIT, that they thought it had merit, and that it would be difficult or impossible to convince people to use it in the real world. We were not surprised at what we view as a reserved (as opposed to enthusiastic) response. Serious objections can be raised in two areas:

- Development time (and expense)
- Resource consumption.

There can be no question that implementing Software BIT requires the involvement of the software engineer during the design and implementation phases. However, we believe that use of a tool (as described in section 3.4.3) will minimize the development time required so that project management personnel will be comfortable with the effort. The remaining technical challenges, described in Chapter 4, can be quantified, demonstrated and evaluated in sufficient time to meet SDI requirements.

Resource consumption is another issue. Software BIT will consume both system memory and processor throughput resources. Additionally, it will use communications assets that might otherwise be allocated to different taskings. Therefore, it is necessary to address the resource issue directly.

Our Phase I proposal included a proposed attempt to identify a potential application within the SDI laboratory environment that could be a target for an experiment demonstrating both the feasibility and benefit associated with Software BIT. Unfortunately, we were unable to find such a potential host until very late in Phase I -- too late to incorporate an experiment design in the material reported herein. (Our effort was, therefore, applied to other aspects of the research.) The Advanced Research Center (ARC) being constructed by the Army's Strategic Defense Command at contractor facilities in the Huntsville area appears to be a very promising candidate. Nevertheless, we feel that a successful laboratory demonstration is crucial to provide the counterbalancing benefits needed to justify the resource utilization of Software BIT.

3.5 LANGUAGE CONSIDERATIONS

Although the generic concept of software BIT does not require any particular computer language, it is reasonable to expect that implementation of the concept may be facilitated by the choice of one or more particular languages. Within the context of SDI, the Ada programming language is clearly the most interesting.

During the Phase I effort, Atlantis investigated two aspects of the Ada language that appeared to offer significant potential within the BIT context: sub-range checking and exception handling. Both are design features of the language that improve the software engineering aspects of Ada code. Our preliminary results are reported in the Appendix and are summarized as follows.

Neither design feature appears to be appropriate for routine application to Software BIT because they tend to interfere with the control flow of the program. As described in section 3.1 above, it does not appear desireable to use a direct intervention strategy except in rare circumstances. Therefore, we conclude that these particular Ada features do not facilitate implementing Software BIT.

Ada remains a strong candidate for application of the concept. It's strong-typing and block structured nature facilitate the implementation of a pre-processor capable of automating the software instrumentation process. Ada's tasking structure is a natural for implementation of both the rate sampling and communications processing tasks that must reside on each processing node. Moreover, the inter-task communications features of Ada may enhance implementation. However, the actual implementation details of particular compilers will significantly influence the design of the Software BIT tools, and it will be necessary to evaluate individual compiler-operating system pairs to determine specific implementation tradeoffs.

4. Conclusions and Recommendations

4.1 PHASE I CONCLUSIONS

Software built-in-test (BIT) is a design technique for collecting information from operational software that will assist in identifying differences between the real Operating Environment and either the Design or Test Environments. The BIT senses and indicates where the software is operating in "new" or "overloaded" environmental conditions and may, therefore, be more likely to fail. (This anomalous situation may be the result of either hardware failure or software design error.) The technical challenge is to incorporate the large number of relatively simple BIT tests into the fault-tolerant and continuously operating environment likely to characterize a solution to the battle management portion of the SDI mission. The management challenge is to provide these technical assists in such a way that they can be implemented in operational software with a minimal increase in software development time; it is then reasonable to expect that BIT will not shift from a hard requirement to a nice-to-have feature as schedule pressures potentially impact development. Our approach overcomes the management problem by providing a standard set of tools for use within the software development environment which will implement BIT with a minimum amount of programmer action.

Four general types of Software BIT appear to provide all of the characteristics needed for both computer-specific and application-specific BIT as defined in our original proposal. These four categories, distilled from numerous candidates, are: event counters, rate samplers, range checkers, and data samplers. They are implemented as pre-processor instructions so that the programmer need only identify the variable(s) to be monitored, type of BIT, limits and priorities at the instrumentation location. The tool then produces the necessary source-code (which implements the BIT, interacts with the BIT database, prepares and dispatches messages, and resumes program execution).

Within the continuously operating, fault-tolerant environment postulated for the SDI battle management application, Software BIT takes the form of three major

components which interact by passing messages through the battle management communications channels. These components are:

- Instrumented software
- Reporting software
- Response software.

Instrumented software is built by designers who fill in templates to provide the parameters which instantiate the generic BIT modules. Reporting software interacts with both the generic modules and the operating system to communicate BIT indications to the on-line maintenance manager. Reporting software will be implemented by project tool builders who provide the necessary "utilities" for use by the instrumented software. Likewise, the response software will be developed on a project basis and may be unique to each major system segment.

It is possible to avoid one of the major pitfalls of automatic test software (i.e., incorrectly set thresholds) using a "training" algorithm during testing and non-crisis operation. In this approach, design threshold values are updated based on system experience. As long as performance remains acceptable, the thresholds are adjusted to reduce false alarms, and the BIT becomes more sensitive to actual environmental changes.

4.2 RECOMMENDATIONS

We offer the following recommendations for additional effort based on the result of our Phase I study:

- Continue the feasibility investigation by designing and prototyping a pre-processor version of the tool.

- Design an experiment for an SDI laboratory demonstration that can demonstrate (or disprove) the utility of the BIT concept. The ARC laboratory in Huntsville represents an excellent candidate based on our limited information received late in the Phase I contract.

- Investigate the implementation trade-offs involved in applying the Software BIT concept to Ada software using at least one second generation Ada compiler.

We believe that a Phase II SBIR effort is an appropriate vehicle for accomplishing these recommendations.

Successful completion of this research will provide a family of tools for assisting the battle manager in anticipating situations where system overload caused by design inconsistency is about to impact performance. This approach should provide a method for precluding or alleviating significant operational impact from a software fault that might otherwise cause anomalous system performance.

Appendix. Using the Exception Handling Features of Ada for Software BIT

Ada contains a construct known as an **Exception**. The LRM refers to them as "facilities for dealing with errors or other exceptional situations that arise during program execution". Exceptions provide a method for changing the normal control flow of the program to handle errors under program control. There are five pre-defined exceptions within Ada, and the user can define additional exceptions. The pre-defined exception **constraint_error** is relevant to these discussions.

Implicit Exceptions. Ada is a strongly typed language which permits range restrictions. Therefore, it is possible to define a variable so that the range feature effectively implements a threshold test; e.g.,

```
subtype Stack_size is INTEGER range min_accept .. max_accept;
```

defines an integer variable Stack_size constrained to the closed interval [min_accept, max_accept]. Any attempt to assign a value to the variable Stack_size outside that limit "**raises**" the exception **constraint_error**.

This approach provides one method of implementing Software BIT. It is both undesirable (in the sense of disturbing the control flow -- see below) and impractical in that it does not provide a useful clue as to the problem. Once **constraint_error** is raised, the program begins its exception handling process. It is unlikely that the exception handler will recognize that it has been invoked by a properly implemented Software BIT function rather than as a result of an error caused by the program. Finally, such an implementation is extremely vulnerable to use of **Pragma SUPPRESS** which can eliminate range checking in order to enhance execution speed.

Explicit Exceptions. It is also possible to declare and handle new types of exceptions. Consider the example presented by the following pseudo-code fragment:

```
declare
     Stack_overflow, Stack_underflow: exception;     -- Example of stack
begin                                                -- handling exceptions
     . . .
     if Stack_size > max_accept then
          raise Stack_overflow;
     elsif Stack_size < min_accept then
          raise Stack_underflow;
     end if;
     . . .
exception
     when Stack_underflow  Stack_overflow =>
          -- handle the stack exceptions
     when others => raise;
end;
```

This example illustrates several points. First, the explicitly defined exceptions Stack_overflow and Stack_underflow act much the same as constraint_error except that the user must explicitly program the conditions under which an exception is raised. Second, the exception handler can process the exception independently or as a related set without interfering with other exceptions. (As written, the **when others** clause provides a mechanism for passing the exception information about other exceptions outside this block. Notice that it is not necessary to identify the type of exception.) Third, because the stack exceptions are processed within the block, they are cleared upon exit (unless they are passed on by raising them again) and will not influence control flow in other portions of the program. Finally, the testing can be moved to another program construct (e.g., **Task** or **Function**) provided that the naming and scope conventions are handled properly.

Using explicit exceptions as a means of implementing Software BIT is one option that warrants further investigation. Although the implied overhead and control interruption penalties may be severe using this approach, they can be controlled using appropriate design techniques. It is not clear whether the disadvantages of this type of implementation outweigh potential benefits.

Control-flow Interruption. Exceptions interfere with the normal flow of program execution. If an exception is raised within a block, program control is passed to the exception handler for that block. If no exception handler exists, control flow passes to the exception handler at the next higher level, bypassing

more and more of the normal program flow until: an exception handler is encountered, the task "is completed" by the exception and a **Tasking_error** exception may be raised, or main program execution is abandoned (which results in an "undefined" state of the system).[5]

The interruption of normal program control-flow in conjunction with exception handling is inconsistent with the philosophy of Software BIT under most circumstances. (Recall that Software BIT is designed to gather information about program performance without interfering with normal program execution on the assumption that the BIT is only intended to gather indirect information. If direct tests of performance are available, Atlantis recommends handling them as part of the design process.) As a result, careful attention must be devoted to scope, blocking, and recovery in order to use exceptions as Software BIT.

---

[5] Although this discussion of exception propagation is technically correct, it is not complete. The interested reader is encouraged to consult either Chapter 11 of the LRM or a suitable text on Ada (e.g., Ref. 5).

References

1. Baxter, A. P. et al, "System 75: Communications and Control Architecture", AT&T Technical Journal, Vol. 64, No. 1, January 1985, pp. 153-173.

2. Sager, G. R. et al, "System 75: The Oryx/Pecos Operating System", Ibid, pp. 251-267.

3. Lu, K. S. et al, "System 75: Maintenance Architecture", Ibid, pp. 229-249.

4. ANSI/MIL-STD-1815A, "Ada Programming Language", 22 Jan 83.

5. Barnes, J.G.P., Programming in Ada, Addison-Wesley (International Series), London, 1982.

6. Siewiorek, Daniel P., "Architecture of Fault-Tolerant Computers", IEEE Computer, Aug 1984, pp 9-18.

7. Wallace, John J. and Walter W. Barnes, "Designing for Ultrahigh Availability: The Unix RTR Operating System", IEEE Computer, Aug 1984, pp 31-39.

Initial Distribution

	Number of Copies
Scientific Officer Office of Naval Research Department of the Navy 800 N. Quincy St. Arlington, VA 22217-5000	3
DCAS - Boston 495 Summer St. Boston, MA 02210-2184	1
Director Naval Research Laboratory Attn: Code 2627 Washington, D.C. 20375	1
Defense Technical Information Center Bldg. 5, Cameron Station Alexandria, VA 22314	12